Introdução à Linguagem

PROLOG

Silvio Lago

Sumário

Capítulo 1
Elementos Básicos

Os elementos básicos da linguagem Prolog são herdados da lógica de predicados. Esses elementos são *fatos*, *regras* e *consultas*.

1.1 Fatos

Fatos servem para estabelecer um relacionamento existente entre objetos de um determinado contexto de discurso. Por exemplo, num contexto bíblico,

```
pai(adão,cain).
```

é um fato que estabelece que Adão é pai de Cain, ou seja, que a relação `pai` existe entre os objetos denominados `adão` e `cain`. Em Prolog, identificadores de relacionamentos são denominados *predicados* e identificadores de objetos são denominados *átomos*. Tanto predicados quanto átomos devem iniciar com letra minúscula.

Programa 1.1: *Uma árvore genealógica.*

```
pai(adão,cain).
pai(adão,abel).
pai(adão,seth).
pai(seth,enos).
```

1.2 Consultas

Para recuperar informações de um programa lógico, usamos *consultas*. Uma consulta pergunta se uma determinado relacionamento existe entre objetos. Por exemplo, a consulta

```
?- pai(adão,cain).
```

pergunta se a relação `pai` vale para os objetos `adão` e `cain` ou, em outras palavras, pergunta se Adão é pai de Cain. Então, dados os fatos estabelecidos no Programa 1.1, a resposta a essa consulta será `yes`. Sintaticamente, fatos e consultas são muito similares. A diferença é que fatos são agrupados no arquivo que constitui o programa, enquanto consultas são sentenças digitadas no *prompt* (`?-`) do interpretador Prolog.

Responder uma consulta com relação a um determinado programa corresponde a determinar se a consulta é *conseqüência lógica* desse programa, ou seja, se a consulta pode ser deduzida dos fatos expressos no programa.

Outra consulta que poderíamos fazer com relação ao Programa 1.1 é

```
?- pai(adão,enos).
```

Nesse caso, porém, o sistema responderia `no`.

As consultas tornam-se ainda mais interessantes quando empregamos *variáveis*, ou seja, identificadores para objetos não especificados. Por exemplo,

```
?- pai(X,abel).
```

pergunta *quem é o pai de Abel* ou, tecnicamente, que valor de `X` torna a consulta uma conseqüência lógica do programa. A essa pergunta o sistema responderá `X = adão`. Note que variáveis devem iniciar com maiúscula.

Uma consulta com variáveis pode ter mais de uma resposta. Nesse caso, o sistema apresentará a primeira resposta e ficará aguardando até que seja pressionado *enter*, que termina a consulta, ou *ponto-e-vírgula*, que faz com que a próxima resposta possível, se houver, seja apresentada.

```
?- pai(adão,X).
X = cain ;
X = abel ;
X = seth ;
no
```

1.2.1 Variável compartilhada

Suponha que desejássemos consultar o Programa 1.1 para descobrir quem é o avô de Enos. Nesse caso, como a relação `avô` não foi diretamente definida nesse programa, teríamos que fazer a seguinte pergunta:

Quem é o pai do pai de Enos?

Então, como o pai de Enos não é conhecido *a priori*, a consulta correspondente a essa pergunta tem dois *objetivos*:

- primeiro, descobrir quem é o pai de Enos, digamos que seja `Y`;
- depois, descobrir quem é o pai de `Y`.

```
?- pai(Y,enos), pai(X,Y).
Y = seth
X = adão
yes
```

Para responder essa consulta, primeiro o sistema resolve `pai(Y,enos)`, obtendo a resposta `Y = seth`. Em seguida, substituindo `Y` por `seth` no segundo objetivo, o sistema resolve `pai(X,seth)`, obtendo `X = adão`.

Nessa consulta, dizemos que a variável `Y` é *compartilhada* pelos objetivos `pai(Y,enos)` e `pai(X,Y)`. Variáveis compartilhadas são úteis porque nos permitem estabelecer restrições entre objetivos distintos.

1.2.2 Variável anônima

Outro tipo de variável importante é a variável *anônima*. Uma variável anônima deve ser usada quando seu valor específico for irrelevante numa determinada consulta. Por exemplo, considerando o Programa 1.1, suponha que desejássemos saber quem já procriou, ou seja, quem tem filhos. Então, como o nome dos filhos é uma informação irrelevante, poderíamos digitar:

```
?- pai(X,_).
```

A essa consulta o sistema responderia `X = adão` e `X = seth`.

1.3 Regras

Regras nos permitem definir novas relações em termos de outras relações já existentes. Por exemplo, a regra

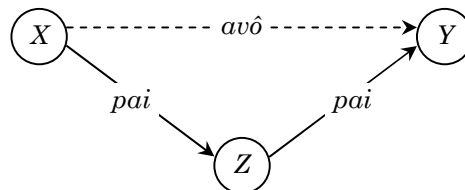
```
avô(X,Y) :- pai(X,Z), pai(Z,Y).
```

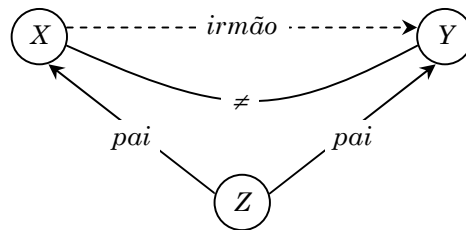
define a relação `avô` em termos da relação `pai`, ou seja, estabelece que `X` é avô de `Y` *se* `X` tem um filho `Z` que é pai de `Y`. Com essa regra, podemos agora realizar consultas tais como

```
?- avô(X,enos).
X = adão
```

Fatos e regras são tipos de *cláusulas* e um conjunto de cláusulas constitui um *programa lógico*.

1.3.1 Grafos de relacionamentos

Regras podem ser formuladas mais facilmente se desenharmos antes um *grafo de relacionamentos*. Nesse tipo de grafo, objetos são representados por *nós* e relacionamentos são representados por *arcos*. Além disso, o arco que representa a relação que está sendo definida deve ser pontilhado. Por exemplo, o grafo a seguir define a relação avô.

A vantagem em se empregar os grafos de relacionamentos é que eles nos permitem visualizar melhor os relacionamentos existentes entre as variáveis usadas numa regra.

Como mais um exemplo, vamos definir a relação `irmão` em termos da relação `pai`, já existente. Podemos dizer que duas pessoas distintas são irmãs se ambas têm o mesmo pai. Essa regra é representada pelo seguinte grafo:

Em Prolog, essa regra é escrita como:

```
irmão(X,Y) :- pai(Z,X), pai(Z,Y), X\=Y.
```

Evidentemente, poderíamos definir a relação `irmão` simplesmente listando todas as suas *instâncias*. Veja:

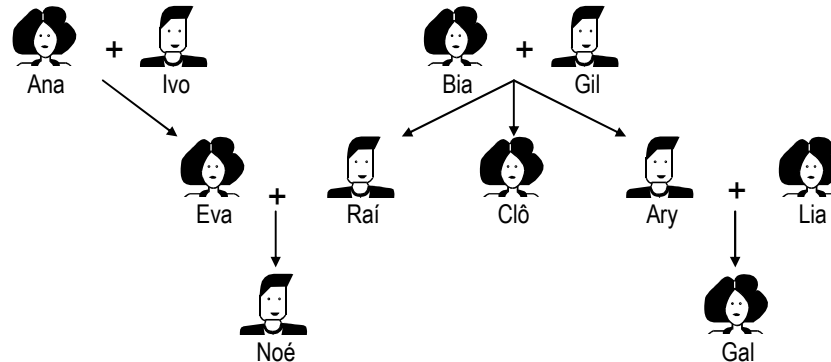
```
irmão(cain,abel).
irmão(cain,seth).
irmão(abel,cain).
irmão(abel,seth).
irmão(seth,cain).
irmão(seth,abel).
```

Entretanto, usar regras, além de muito mais elegante e conciso, também é muito mais consistente. Por exemplo, se o fato `pai(adão,eloi)` fosse acrescentado ao Programa 1.1, usando a definição por regra, nada mais precisaria ser alterado. Por outro lado, usando a definição por fatos, teríamos que acrescentar ao programa mais seis novas instâncias da relação `irmão`.

1.4 Exercícios

1.1. Digite o Programa 1.1, incluindo as regras que definem as relações `avô` e `irmão`, e realize as seguintes consultas:

a) *Quem são os filhos de Adão?*
b) *Quem são os netos de Adão?*
c) *Quem são os tios de Enos?*

1.2. Considere a árvore genealógica a seguir:

a) Usando *fatos*, defina as relações `pai` e `mãe`. Em seguida, consulte o sistema para ver se suas definições estão corretas.

b) Acrescente ao programa os fatos necessários para definir as relações `homem` e `mulher`. Por exemplo, para estabelecer que Ana é mulher e Ivo é homem, acrescente os fatos `mulher(ana)` e `homem(ivo)`.

c) Usando duas regras, defina a relação `gerou(X,Y)` tal que `X` gerou `Y` se `X` é pai ou mãe de `Y`. Faça consultas para verificar se sua definição está correta. Por exemplo, para a consulta `gerou(X,eva)` o sistema deverá apresentar as respostas `X = ana` e `X = ivo`.

d) Usando relações já existentes, crie regras para definir as relações `filho`, `filha`, `tio`, `tia`, `primo`, `prima`, `avô` e `avó`. Para cada relação, desenhe o grafo de relacionamentos, codifique a regra correspondente e faça consultas para verificar a corretude.

1.3. Codifique as regras equivalentes às seguintes sentenças:

a) *Todo mundo que tem filhos é feliz.*

b) *Um casal é formado por duas pessoas que têm filhos em comum.*

Capítulo 2
Banco de Dados Dedutivo

Um conjunto de fatos e regras definem um banco de dados dedutivo, com a mesma funcionalidade de um banco de dados relacional.

2.1 Aritmética

Prolog oferece um predicado especial `is`, bem como um conjunto *operadores*, através dos quais podemos efetuar operações aritméticas.

```
?- X is 2+3.

X = 5
```

Os operadores aritméticos são `+` (*adição*), `-` (*subtração*), `*` (*multiplicação*), `mod` (*resto*), `/` (*divisão real*), `//` (*divisão inteira*) e `^` (*potenciação*).

Programa 2.1: *Área e população dos países.*

```
% país(Nome, Área, População)

  país(brasil,  9,  130).
  país(china,  12, 1800).
  país(eua,     9,  230).
  país(índia,   3,  450).
```

O Programa 2.1 representa uma tabela que relaciona a cada país sua área em Km^2 e sua população em milhões de habitantes. Note que a linha iniciando com `%` é um comentário e serve apenas para fins de documentação. Com base nesse programa, por exemplo, podemos determinar a densidade demográfica do Brasil, através da seguinte consulta:

```
?- país(brasil,A,P), D is P/A.
A = 9
P = 130
D = 14.4444
```

Uma outra consulta que poderia ser feita é a seguinte: "*Qual a diferença entre a população da China e da Índia?*".

```
?- país(china,_,X), país(índia,_,Y), Z is X-Y.
X = 1800
Y = 450
Z = 1350
```

2.2 Comparação

Para realizar comparações numéricas podemos usar os seguintes predicados primitivos: `=:=` (*igual*) , `=\=` (*diferente*), `>` (*maior*), `>=` (*maior ou igual*), `<` (*menor*) e `=<` (*menor ou igual*). Por exemplo, com esses predicados, podemos realizar consultas tais como:

- *A área do Brasil é igual à área dos Estados Unidos?*

```
?- país(brasil,X,_), país(eua,Y,_), X =:= Y.
X = 9
Y = 9
Yes
```

- *A população dos Estados Unidos é maior que a população da Índia?*

```
?- país(eua,_,X), país(índia,_,Y), X > Y.
No
```

2.3 Relacionamentos entre tabelas

O quadro a seguir relaciona a cada funcionário de uma empresa seu *código*, seu *salário* e os seus *dependentes*.

Código	Nome	Salário	Dependentes
1	Ana	R$ 1000,90	Ary
2	Bia	R$ 1200,00	------
3	Ivo	R$ 903,50	Raí, Eva

Usando os princípios de modelagem lógica de dados (1FN), podemos representar as informações desse quadro através do uso de duas tabelas: a primeira contendo informações sobre os funcionários e a segunda contendo informações sobre os dependentes. Em Prolog, essas tabelas podem ser representadas por meio de dois predicados distintos: `func` e `dep`.

Programa 2.2: *Funcionários e dependentes.*

```
% func(Código, Nome, Salário)
  func(1, ana, 1000.90).
  func(2, bia, 1200.00).
  func(3, ivo,  903.50).

% dep(Código, Nome)
  dep(1, ary).
  dep(3, raí).
  dep(3, eva).
```

Agora, com base no Programa 2.2, podemos, por exemplo, consultar o sistema para recuperar os dependentes de Ivo, veja:

```
?- func(C,ivo,_), dep(C,N).
C = 3
N = raí ;
C = 3
N = eva
```

Observe que nessa consulta, o campo *chave* `C` é uma variável compartilhada. É graças a essa variável que o relacionamento entre funcionário e dependentes, existente na tabela original, é restabelecido.

Outra coisa que podemos fazer é descobrir de quem Ary é dependente:

```
?- dep(C,ary), func(C,N,_).
C = 1
N = ana
```

Ou, descobrir quem depende de funcionário com salário inferior a R$ 950,00:

```
?- func(C,_,S), dep(C,N), S<950.
C = 3
S = 903.5
N = raí ;
C = 3
S = 903.5
N = eva
```

Finalmente, poderíamos também consultar o sistema para encontrar funcionários que não têm dependentes:

```
?- func(C,N,_), not dep(C,_).
C = 2
N = bia
```

Nessa última consulta, `not` é um predicado primitivo do sistema Prolog que serve como um tipo especial de negação, denominada *negação por falha*, que será estudada mais adiante. Por enquanto, é suficiente saber que o predicado `not` só funciona apropriadamente quando as variáveis existentes no objetivo negado já se encontram instanciadas no momento em que o predicado é avaliado. Por exemplo, na consulta acima, para chegar ao objetivo `not dep(C,_)`, primeiro o sistema precisa resolver o objetivo `func(C,N,_)`; mas, nesse caso, ao atingir o segundo objetivo, a variável `C` já foi substituída por uma constante (no caso, o número `2`).

2.4 O modelo de dados relacional

Programas lógicos são uma poderosa extensão do modelo de dados relacional. Conjuntos de fatos correspondem às tabelas do modelo relacional e as operações básicas da álgebra relacional (*seleção*, *projeção, união*, *diferença simétrica* e *produto cartesiano*) podem ser facilmente implementadas através de regras. Como exemplo, considere o Programa 2.3.

Programa 2.3: *Uma tabela de filmes.*

```
% filme(Título, Gênero, Ano, Duração)
  filme('Uma linda mulher', romance,  1990, 119).
  filme('Sexto sentido',    suspense, 2001, 108).
  filme('A cor púrpura',    drama,    1985, 152).
  filme('Copacabana',       comédia,  2001,  92).
  filme('E o vento levou',  drama,    1939, 233).
  filme('Carrington',       romance,  1995, 130).
```

Suponha que uma locadora precisasse de uma tabela contendo apenas filmes clássicos (*i.e.* lançados até 1985), para uma determinada promoção. Então, teríamos que realizar uma *seleção* na tabela de filmes:

```
clássico(T,G,A,D) :- filme(T,G,A,D), A =< 1985.
```

Suponha ainda que a locadora desejasse apenas os nomes e os gêneros dos filmes clássicos. Nesse caso, teríamos que usar também *projeção*:

```
clássico(T,G) :- filme(T,G,A,_), A =< 1985.
```

Agora, fazendo uma consulta com esse novo predicado `clássico`, obteríamos as seguintes respostas:

```
?- clássico(T,G).
T = 'A cor púrpura'
G = drama ;
T = 'E o vento levou'
G = drama
```

2.5 Exercícios

2.1. Inclua no Programa 2.1 uma regra para o predicado `dens(P,D)`, que relaciona cada país `P` à sua densidade demográfica correspondente `D`. Em seguida, faça consultas para descobrir:

a) *qual a densidade demográfica de cada um dos países;*
b) *se a Índia é mais populosa que a China.*

2.2. Inclua no Programa 2.2 as informações da tabela abaixo e faça as consultas indicadas a seguir:

Código	Nome	Salário	Dependentes
4	Leo	R$ 2500,35	Lia, Noé
5	Clô	R$ 1800,00	Eli
6	Gil	R$ 1100,00	------

a) *Quem tem salário entre R$ 1500,00 e R$ 3000,00?*
b) *Quem não tem dependentes e ganha menos de R$ 1200,00?*
c) *Quem depende de funcionário que ganha mais de R$ 1700,00?*

2.3. Inclua no Programa 2.3 as seguintes regras:

a) *Um filme é longo se tem duração superior a 150 minutos.*
b) *Um filme é lançamento se foi lançado a menos de 1 ano.*

2.4. Codifique um programa contendo as informações da tabela abaixo e faça as consultas indicadas a seguir:

Nome	Sexo	Idade	Altura	Peso
Ana	fem	23	1.55	56.0
Bia	fem	19	1.71	61.3
Ivo	masc	22	1.80	70.5
Lia	fem	17	1.85	57.3
Eva	fem	28	1.75	68.7
Ary	masc	25	1.72	68.9

a) *Quais são as mulheres com mais de 20 anos de idade?*
b) *Quem tem pelo menos 1.70m de altura e menos de 65kg?*
c) *Quais são os possíveis casais onde o homem é mais alto que a mulher?*

2.5. O peso ideal para uma modelo é no máximo `62.1*Altura-44.7` . Além disso, para ser modelo, uma mulher precisa ter mais que 1.70m de altura e menos de 25 anos de idade. Com base nessas informações, e considerando a tabela do exercício anterior, defina um predicado capaz de recuperar apenas os nomes das mulheres que podem ser modelos.

Capítulo 3
Controle Procedimental

Embora Prolog seja uma linguagem essencialmente declarativa, ela provê recursos que nos permitem interferir no comportamento dos programas.

3.1 Retrocesso

O núcleo do Prolog, denominado *motor de inferência*, é a parte do sistema que implementa a estratégia de busca de soluções. Ao satisfazer um objetivo, geralmente há mais de uma cláusula no programa que pode ser empregada; como apenas uma delas pode ser usada de cada vez, o motor de inferência seleciona a primeira delas e reserva as demais para uso futuro.

Programa 3.1: *Números binários de três dígitos.*

```
d(0).                                          % cláusula 1
d(1).                                          % cláusula 2
b([A,B,C]) :- d(A), d(B), d(C).                % cláusula 3
```

Por exemplo, para satisfazer o objetivo

```
?- b(N).
```

a única opção que o motor de inferência tem é selecionar a terceira cláusula do Programa 3.1. Então, usando um mecanismo denominado *resolução*, o sistema fará `N=[A,B,C]` e reduzirá[1] a consulta inicial a uma nova consulta:

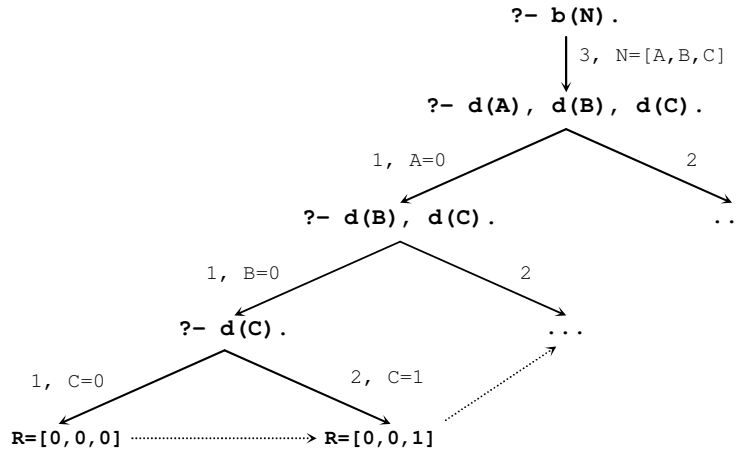
```
?- d(A), d(B), d(C).
```

Quando uma consulta contém vários objetivos, o sistema seleciona sempre aquele mais à esquerda e, portanto, o próximo objetivo a ser resolvido será `d(A)`. Para resolver `d(A)`, há duas opções: cláusulas 1 e 2. O sistema selecionará a primeira delas, deixando a outra para depois. Usando a cláusula 1, ficaremos, então, com `A=0`, `N=[0,B,C]` e a consulta será reduzida a:

```
?- d(B), d(C).
```

A partir daí, os próximos dois objetivos serão resolvidos analogamente a `d(A)` e a resposta `N=[0,0,0]` será exibida no vídeo. Nesse ponto, se a tecla *ponto-e-vírgula* for pressionada, o mecanismo de *retrocesso* será acionado e o sistema tentará encontrar uma resposta alternativa. Para tanto, o motor de inferência retrocederá na última escolha feita e selecionará a próxima alter-

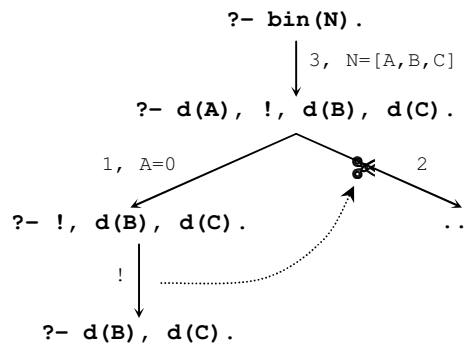
[1] *Reduzir significa que um objetivo complexo é substituído por objetivos mais simples.*

nativa. A figura a seguir mostra a árvore de busca construída pelo motor de inferência, até esse momento, bem como o resultado do retrocesso.

3.2 Cortes

Nem sempre desejamos que todas as possíveis respostas a uma consulta sejam encontradas. Nesse caso, podemos instruir o sistema a *podar* os ramos indesejáveis da árvore de busca e ganhar eficiência.

Tomando como exemplo o Programa 3.1, para descartar os números binários iniciando com o dígito 1, bastaria podar o ramo da árvore de busca que faz `A=1`. Isso pode ser feito do seguinte modo:

```
bin([A,B,C]) :- d(A), !, d(B), d(C).
```

A execução do predicado `!` (*corte*) poda todos os ramos ainda não explorados, a partir do ponto em que a cláusula com o corte foi selecionada.

3.2.1 Evitando retrocesso desnecessário

Vamos analisar um exemplo cuja execução envolve retrocesso desnecessário e veremos como evitar este problema com o uso de corte. Considere a função:

$$f(x) = \begin{cases} 0 \text{ se } x < 5 \\ 1 \text{ se } x \geq 5 \text{ e } x \leq 9 \\ 2 \text{ se } x > 9 \end{cases}$$

Essa função pode ser codificada em Prolog da seguinte maneira:

Programa 3.2: *Uma função matemática.*

```
f(X,0) :- X<5.                                    % cláusula 1
f(X,1) :- X>=5, X=<9.                             % cláusula 2
f(X,2) :- X>9.                                    % cláusula 3
```

Para enxergar onde há retrocesso desnecessário, considere a consulta

```
?- f(1,Y).
```

Para resolver `f(1,Y)`, primeiro selecionamos a cláusula 1. Com ela, obtemos `X=1`, `Y=0` e a consulta é reduzida ao objetivo `1<5`. Como esse objetivo é verdadeiro, a resposta `Y=0` é exibida no vídeo. Então, caso o usuário pressione *ponto-e-vírgula*, o retrocesso selecionará a cláusula 2. Isto resultará em novos valores para `X` e `Y` e a consulta será reduzida aos objetivos `1>=5` e `1=<9`. Como esses objetivos não podem ser satisfeitos, o retrocesso será acionado automaticamente e a cláusula 3 será selecionada. Com essa última cláusula, a consulta será reduzida a `1>5` e, como esse objetivo também não pode ser satisfeito, o sistema exibirá a resposta `no`.

De fato, as três regras que definem a função *f(x)* são mutuamente exclusivas e, portanto, apenas uma delas terá sucesso para cada valor de *x*. Para evitar que após o sucesso de uma regra, outra deltas seja inutilmente selecionada, podemos usar cortes. Quando o motor de inferência seleciona uma cláusula e executa um corte, automaticamente, ele descarta todas as demais cláusulas existentes para o predicado em questão. Dessa forma, evitamos retrocesso desnecessário e ganhamos velocidade de execução.

Programa 3.3: *Uso de cortes para evitar retrocesso desnecessário.*

```
f(X,0) :- X<5, !.                                 % cláusula 1
f(X,1) :- X>=5, X=<9, !.                          % cláusula 2
f(X,2) :- X>9.                                    % cláusula 3
```

3.2.2 Estrutura condicional

Embora não seja considerado um estilo declarativo puro, é possível criar em Prolog um predicado para implementar a estrutura condicional *if-then-else*.

Programa 3.4: *Comando if-then-else.*

```
if(Condition,Then,Else) :- Condition, !, Then.
if(_,_,Else) :- Else.
```

Para entender como esse predicado funciona, considere a consulta a seguir:

```
?- if(8 mod 2 =:= 0, write(par), write(ímpar)).
```

Usando a primeira cláusula do Programa 3.4, obtemos

```
Condition = 8 mod 2 =:= 0
Then = write(par)
Else = write(ímpar)
```

e a consulta é reduzida a três objetivos:

```
?- 8 mod 2 =:= 0,  !, write(par).
```

Como a condição expressa pelo primeiro objetivo é verdadeira, mais uma redução é feita pelo sistema e obtemos

```
?- !, write(par).
```

Agora o corte é executado, fazendo com que a segunda cláusula do programa seja descartada, e a consulta torna-se

```
?- write(par).
```

Finalmente, executando-se `write`, a palavra `par` é exibida no vídeo e o processo termina.

Considere agora essa outra consulta:

```
?- if(5 mod 2 =:= 0, write(par), write(ímpar)).
```

Novamente a primeira cláusula é selecionada e obtemos

```
?- 8 mod 2 =:= 0,  !, write(par).
```

Nesse caso, porém, como a condição expressa pelo primeiro objetivo é falsa, o corte não chega a ser executado e a segunda cláusula do programa é, então, selecionada pelo retrocesso. Como resultado da seleção dessa segunda cláusula, a palavra `ímpar` é exibida no vídeo e o processo termina.

3.3 Falhas

Vamos retomar o Programa 3.1 como exemplo:

```
?- b(N).
N = [0,0,0] ;
N = [0,0,1] ;
N = [0,1,0] ;
...
```

Podemos observar na consulta acima que, a cada resposta exibida, o sistema fica aguardando o usuário pressionar a tecla `';'` para buscar outra solução.

Uma forma de forçar o retrocesso em busca de soluções alternativas, sem que o usuário tenha que solicitar, é fazer com que após cada resposta obtida o sistema encontre um objetivo *insatisfatível*. Em Prolog, esse objetivo é representado pelo predicado `fail`, cuja execução sempre provoca uma falha.

Programa 3.5: *Uso de falhas para recuperar respostas alternativas.*

```
d(0).
d(1).
bin :- d(A), d(B), d(C), write([A,B,C]), nl, fail.
```

O Programa 3.5 mostra como podemos usar o predicado `fail`. A execução dessa versão modificada do Programa 3.1 exibirá todas as respostas, sem interrupção, da primeira até a última:

```
?- bin.
[0,0,0]
[0,0,1]
[0,1,0]
...
```

3.4 Exercícios

3.1. O programa a seguir associa a cada pessoa seu esporte preferido.

```
joga(ana,volei).
joga(bia,tenis).
joga(ivo,basquete).
joga(eva,volei).
joga(leo,tenis).
```

Suponha que desejamos consultar esse programa para encontrar um parceiro `P` para jogar com Leo. Então, podemos realizar essa consulta de duas formas:

a) `?- joga(P,X), joga(leo,X), P\=leo.`
b) `?- joga(leo,X), joga(P,X), P\=leo.`

Desenhe as árvores de busca construídas pelo sistema ao responder cada uma dessas consultas. Qual consulta é mais eficiente, por quê?

3.2. O predicado `num` classifica números em três categorias: *positivos*, *nulo* e *negativos*. Esse predicado, da maneira como está definido, realiza retrocesso desnecessário. Explique por que isso acontece e, em seguida, utilize cortes para eliminar esse retrocesso.

```
num(N,positivo) :- N>0.
num(0,nulo).
num(N,negativo) :- N<0.
```

3.3. Suponha que o predicado `fail` não existisse em Prolog. Qual das duas definições a seguir poderia ser corretamente usada para causar falhas?

a) `falha :- (1=1).`
b) `falha :- (1=2).`

3.4. Considere o programa a seguir:

```
animal(cão).
animal(canário).
animal(cobra).
animal(morcego).
animal(gaivota).

voa(canário).
voa(morcego).
voa(gaivota).

dif(X,X) :- !, fail.
dif(_,_).

pássaro(X) :- animal(X), voa(X), dif(X,morcego).
```

Desenhe a árvore de busca necessária para responder a consulta

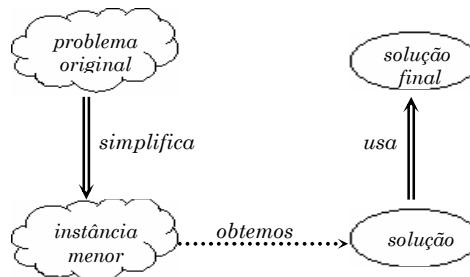
```
?- pássaro(X).
```

Em seguida, execute o programa para ver se as respostas do sistema correspondem àquelas que você encontrou.

Capítulo 4
Programação Recursiva

Recursividade é fundamental em Prolog; graças ao seu uso, programas realmente práticos podem ser implementados.

4.1 Recursividade

A *recursividade* é um princípio que nos permite obter a solução de um problema a partir da solução de uma instância menor dele mesmo. Para aplicar esse princípio, devemos assumir como hipótese que a solução da instância menor é conhecida. Por exemplo, suponha que desejamos calcular 2^{11}. Uma instância menor desse problema é 2^{10} e, para essa instância, "sabemos" que a solução é 1024. Então, como $2 \times 2^{10} = 2^{11}$, concluímos que $2^{11} = 2 \times 1024 = 2048$.

A figura acima ilustra o princípio de recursividade. De modo geral, procedemos da seguinte maneira: simplificamos o *problema original* transformando-o numa *instância menor*; então, *obtemos* a *solução* para essa instância e a *usamos* para construir a *solução final*, correspondente ao problema original.

O que é difícil de entender, *a priori*, é como a solução para a instância menor é obtida. Porém, não precisamos nos preocupar com essa parte. A solução da instância menor é gerada pelo próprio mecanismo da recursividade. Sendo assim, tudo o que precisamos fazer é encontrar uma simplificação adequada para o problema em questão e descobrir como a solução obtida recursivamente pode ser usada para construir a solução final.

4.2 Predicados recursivos

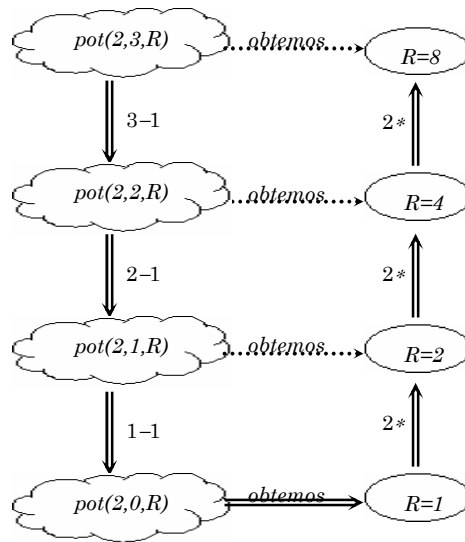
A definição de um *predicado recursivo* é composta por duas partes:

1º *base*: resolve diretamente a instância mais simples do problema.
2º *passo*: resolve instâncias maiores, usando o princípio de recursividade.

Programa 4.1: *Cálculo de potência.*

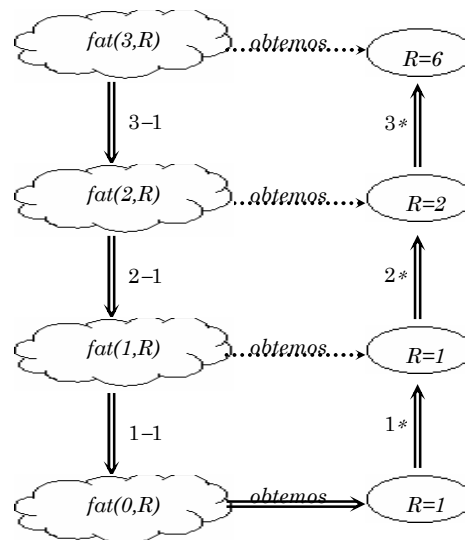
```
% pot(Base,Expoente,Potência)
  pot(_,0,1).     % base
  pot(B,N,P) :-   % passo
      N>0,          % condição do passo
      M is N-1,     % simplifica o problema
      pot(B,M,R),   % obtém solução da instância menor
      P is B*R.     % constrói solução final
```

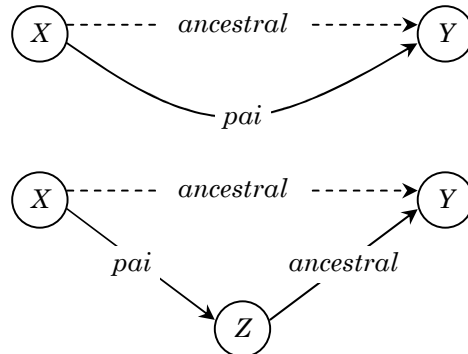
O Programa 4.1 mostra a definição de um predicado para calcular potências. A base para esse problema ocorre quando o expoente é `0`, já que qualquer número elevado a `0` é igual a `1`. Por outro lado, se o expoente é maior que `0`, então o problema deve ser simplificado, ou seja, temos que chegar um pouco mais perto da base. A chamada recursiva com `M` igual a `N-1` garante justamente isso. Portanto, após um número finito de passos, a base do problema é atingida e o resultado esperado é obtido.

Um modo de entender o funcionamento dos predicados recursivos é desenhar o *fluxo de execução*.

A cada expansão, deixamos a oval em branco e rotulamos a seta que sobre com a operação que fica pendente. Quando a base é atingida, começamos a preencher as ovais, propagando os resultados de baixo para cima e efetuando as operações pendentes. O caminho seguido é aquele indicado pelas setas duplas. As setas pontilhadas representam a hipótese de recursividade.

Programa 4.2: *Cálculo de fatorial.*

```
% fat(Número,Fatorial)
  fat(0,1).      % base
  fat(N,F) :-    % passo
      N>0,         % condição do passo
      M is N-1,    % simplifica o problema
      fat(M,R),    % obtém solução da instância menor
      F is N*R.    % constrói solução final
```

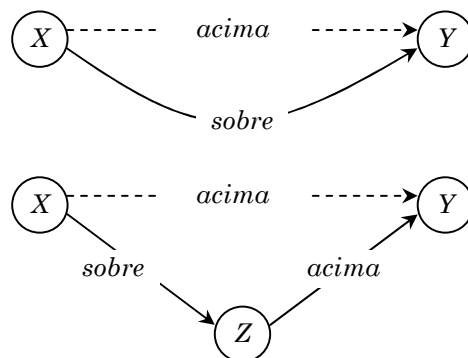
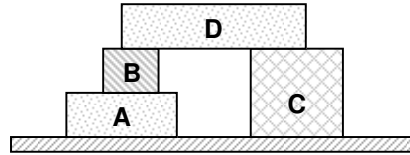
O Programa 4.2 mostra mais um exemplo de predicado recursivo e a figura a seguir mostra o fluxo de execução para a consulta `?- fat(3,R).`

4.3 Relações transitivas

Se uma relação *r* é *transitiva*, então *r*(*x*,*y*) e *r*(*y*,*z*) implicam *r*(*x*,*z*). Um exemplo desse tipo de relação é a relação *ancestral*: se Adão é ancestral de Seth e Seth é ancestral de Enos, então Adão também é ancestral de Enos. Uma relação transitiva é sempre definida em termos de uma outra relação, denominada relação *base*. No caso da relação *ancestral*, a relação base é a relação *pai*. Assim, podemos dizer que se um indivíduo *x* é pai de um indivíduo *y*, então *x* é ancestral de *y*. Além disso, se *x* é pai de *z* e *z* é ancestral de *y*, então *x* também é ancestral de *y*. Essas regras podem ser visualizadas nos grafos de relacionamentos a seguir:

Programa 4.3: *A relação transitiva ancestral.*

```
pai(adão,cain).
pai(adão,abel).
pai(adão,seth).
pai(seth,enos).
ancestral(X,Y) :- pai(X,Y).
ancestral(X,Y) :- pai(X,Z), ancestral(Z,Y).
```

Veja uma consulta que poderia ser feita com o Programa 4.3:

```
?- ancestral(X,enos).
X = seth ;
X = adão
```

Outro exemplo interessante de relação transitiva é a relação *acima*, cuja relação base é a relação *sobre*. Os grafos a seguir descrevem essa relação:

Programa 4.4: *A relação transitiva acima.*

```
sobre(b,a).
sobre(d,b).
sobre(d,c).

acima(X,Y) :- sobre(X,Y).
acima(X,Y) :- sobre(X,Z), acima(Z,Y).
```

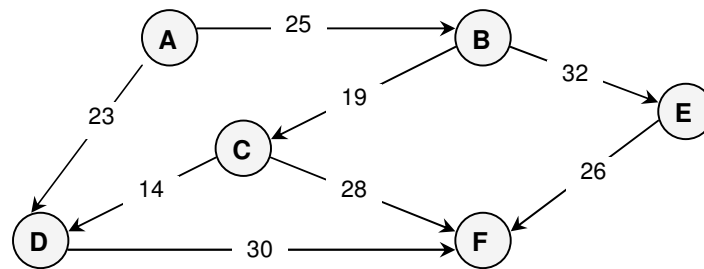
Veja uma consulta que poderia ser feita com o Programa 4.4:

```
?- acima(X,a).
X = b ;
X = d
```

4.4 Exercícios

4.1. Defina um predicado recursivo para calcular o produto de dois números naturais usando apenas soma e subtração.

4.2. Defina um predicado recursivo exibir um número natural em binário.

4.3. O grafo a seguir representa um mapa, cujas cidades são representadas por letras e cujas estradas (de sentido único) são representados por números, que indicam sua extensão em km.

a) Usando o predicado `estrada(Origem,Destino,Km)`, crie um programa para representar esse mapa.

b) Defina a relação transitiva `dist(A,B,D)`, que determina a distância `D` entre duas cidades `A` e `B`.

Capítulo 5
Listas e Estruturas

Listas e estruturas são os dois mecanismos básicos existentes na linguagem Prolog para criação de estruturas de dados mais complexas.

5.1 Listas

Uma *lista* é uma seqüência linear de itens, separados por vírgulas e delimitados por colchetes. A lista *vazia* é representada por `[]` e uma lista com pelo menos um item é representada por `[X|Y]`, onde `X` é a *cabeça* (primeiro item) e `Y` é a *cauda* (demais itens) dessa lista.

```
?- [X|Y] = [terra, sol, lua].
X = terra
Y = [sol, lua]
Yes
?- [X|Y] = [estrela].
X = estrela
Y = []
Yes
?- [X|Y] = [].
No
```

A tabela a seguir lista alguns padrões úteis na manipulação de listas:

Padrão	Quantidade de itens na lista
[]	Nenhum
[X]	um único item
[X|Y]	pelo menos um item
[X, Y]	exatamente dois itens
[X, Y|Z]	pelo menos dois itens
[X, Y, Z]	exatamente três itens

Por exemplo, para selecionar o terceiro item de uma lista podemos fazer:

```
?- [_,_,X|_] = [a,b,c,d,e].
X = c
Yes
```

5.1.1 Tratamento recursivo de listas

Usando o princípio de recursividade podemos percorrer uma lista acessando seus itens, um a um, seqüencialmente.

Programa 5.1: *Exibição de listas.*

```
% exibe(L) : exibe os elementos da lista L
  exibe([]) :- nl.
  exibe([X|Y]) :- write(X), exibe(Y).
```

O Programa 5.1 mostra um predicado para exibição de listas. Quando a lista está vazia, ele apenas muda o cursor para uma nova linha (`nl`). Quando a lista tem pelo menos um item, ele exibe o primeiro deles usando `write` e faz uma chamada recursiva para exibir os demais itens. A cada chamada recursiva a lista terá um item a menos. Portanto, chegará um momento em que a lista ficará vazia e, então, o processo terminará.

Programa 5.2: *Verifica se um item é membro de uma lista.*

```
% membro(X,L) : o item X é membro da lista L
  membro(X,[X|_]).
  membro(X,[_|Y]) :- membro(X,Y).
```

O Programa 5.2 mostra um outro predicado cuja finalidade principal é determinar se um item `X` consta numa lista `L`.

```
?- membro(c,[a,b,c,d]).
Yes

?- membro(e,[a,b,c,d]).
No
```

O interessante é que esse predicado também ser usado para acessar os itens existentes numa lista, veja:

```
?- membro(X,[a,b,c,d]).
X = a ;
X = b ;
X = c ;
X = d
```

Programa 5.3: *Anexa duas listas.*

```
% anexa(A,B,C): A anexado com B dá C
  anexa([], B, B).
  anexa([X|A], B, [X|C]) :- anexa(A, B, C).
```

Outro predicado bastante útil é aquele apresentado no Programa 5.3. Conforme a primeira cláusula estabelece, quando uma lista vazia é anexada a uma lista `B`, o resultado será a própria lista `B`. Caso contrário, se a primeira lista não for vazia, então podemos anexar a sua cauda `A` com a segunda lista `B` e, depois, prefixar sua cabeça `X` ao resultado obtido `C`.

O predicado `anexa` tem como finalidade básica anexar duas listas, mas também pode ser usado de outras formas interessantes.

```
?- anexa([a,b],[c,d],L).
L = [a,b,c,d]

?- anexa([a,b],L, [a,b,c,d]).
L = [c,d]

?- anexa(X,Y,[a,b,c]).
X = []
Y = [a,b,c] ;

X = [a]
Y = [b,c] ;

X = [a, b]
Y = [c] ;

X = [a,b,c]
Y = []
```

5.1.2 Ordenação de listas

Vamos ordenar uma lista `L`, usando um algoritmo conhecido como *ordenação por intercalação*. Esse algoritmo consiste de três passos básicos:

- *distribuir* os itens da lista `L` entre duas sublistas `A` e `B`, de modo que elas tenham tamanhos aproximadamente iguais;
- *ordenar* recursivamente as sublistas `A` e `B`, obtendo-se, respectivamente, as sublistas `As` e `Bs`;
- *intercalar* as sublistas `As` e `Bs`, obtendo-se a lista ordenada `S`.

Programa 5.4: *Algoritmo Merge Sort.*

```
% distribui(L,A,B) : distribui itens de L entre A e B
  distribui([],[],[]).
  distribui([X],[X],[]).
  distribui([X,Y|Z],[X|A],[Y|B]) :- distribui(Z,A,B).
```

```
% intercala(A,B,L) : intercala A e B gerando L
  intercala([],B,B).
  intercala(A,[],A).
  intercala([X|A],[Y|B],[X|C]) :-
    X =< Y,
    intercala(A,[Y|B],C).
  intercala([X|A],[Y|B],[Y|C]) :-
    X > Y,
    intercala([X|A],B,C).

% ordena(L,S) : ordena a lista L obtendo S
  ordena([],[]).
  ordena([X],[X]).
  ordena([X,Y|Z],S) :-
    distribui([X,Y|Z],A,B),
    ordena(A,As),
    ordena(B,Bs),
    intercala(As,Bs,S).
```

Usando o Programa 5.4 podemos fazer a seguinte consulta:

```
?- ordena([3,5,0,4,1,2],S).
S = [0,1,2,3,4,5]
```

Para ordenar a lista `[3,5,0,4,1,2]`, primeiro o predicado distribui é chamado. Como a lista tem mais de um item, a terceira cláusula do predicado `distribui` é utilizada. Essa cláusula remove os dois primeiros itens da lista e distribui os demais itens entre `A` e `B`, recursivamente; em seguida, adiciona um dos itens removidos em `A` e o outro em `B`. Essa política *"um pra mim, um pra você"* é que garante que a distribuição será equilibrada.

```
?- distribui([3,5,0,4,1,2],A,B).
A = [3, 0, 1]
B = [5, 4, 2]
```

Depois que a distribuição é feita, a recursividade se encarrega de ordenar cada uma das sublistas e produzir `As=[0,1,3]` e `Bs=[2,4,5]`, que são passadas como entrada ao predicado `intercala`. Esse predicado, então, compara o primeiro item da lista `As` com o primeiro item da lista `Bs` e seleciona o menor deles. Após intercalar recursivamente os itens restantes, o item selecionado é inserido no início da lista obtida pela intercalação recursiva.

```
?- intercala([0,1,3],[2,4,5],S).
S = [0,1,2,3,4,5]
```

5.2 Estruturas

Estruturas são objetos de dados que possuem uma quantidade fixa de componentes, cada um deles podendo ser acessado individualmente. Por exemplo, `data(6,agosto,2003)` é uma estrutura cujos componentes são `6`, `agosto` e `2003`. Para combinar os componentes de uma estrutura usamos um *functor*. Nesse exemplo, o functor é a palavra `data`. Uma estrutura tem a forma de um fato, mas pode ser usada como argumento de um predicado.

Programa 5.6: *Acontecimentos históricos.*

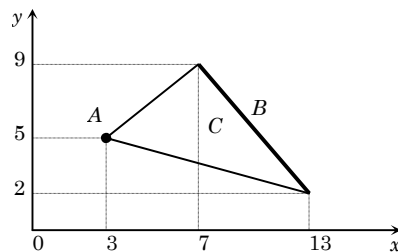
```
hist(data(22,abril,1500), 'Descobrimento do Brasil').
hist(data(7,setembro,1822),'Declaração da independência').
hist(data(15,novembro,1888),'Proclamação da República').
```

As consultas a seguir, feitas com base no Programa 5.6, mostram o uso de estruturas em Prolog:

```
?- hist(data(7,setembro,1822),F).
F = 'Declaração da independência'

?- hist(D,'Proclamação da República').
D = data(15, novembro, 1888)
```

5.1.2 Representando objetos geométricos

Veremos agora como estruturas podem ser empregadas para representar alguns objetos geométricos simples: um ponto será representado por suas coordenadas no plano cartesiano, uma linha será definida por dois pontos, e um triângulo será definido por três pontos:

Usando os functores `ponto`, `linha` e `triângulo`, alguns dos objetos da figura acima podem ser representados do seguinte modo:

```
A = ponto(3,5)
B = linha(ponto(7,9),ponto(13,2))
C = triângulo(ponto(3,5),ponto(7,9),ponto(13,2))
```

Programa 5.6: *Verificando linhas verticais e horizontais.*

```
vertical( linha(ponto(X,_), ponto(X,_)) ).
horizontal( linha(ponto(_,Y), ponto(_,Y)) ).
```

O Programa 5.6 mostra como podemos obter resultados interessantes usando estruturas. Esse programa estabelece que uma linha é vertical se seus extremos têm a mesma ordenada e que ela é horizontal se os seus extremos têm mesma abcissa. Com esse programa, podemos fazer consultas tais como:

```
?- vertical(linha(ponto(1,1),ponto(1,2))).
Yes

?- vertical(linha(ponto(1,1),ponto(2,Y))).
No

?- horizontal(linha(ponto(1,1),ponto(2,Y))).
Y = 1
Yes
```

Ou ainda a consulta: '*Existe linha vertical com extremo em (2,3)?*'

```
?- vertical( linha(ponto(2,3),P) ).
P = ponto(2,_0084)
Yes
```

Cuja resposta exibida pelo sistema significa: '*sim, qualquer linha que começa no ponto (2,3) e termina no ponto (2,_) é uma resposta à questão*'.

Outra consulta interessante seria: '*Existe uma linha vertical e horizontal ao mesmo tempo?*'

```
?- vertical(L), horizontal(L).
L = linha(ponto(_0084,_0085),ponto(_0084,_0085))
Yes
```

Esta resposta significa: '*sim, qualquer linha degenerada a um ponto tem a propriedade de ser vertical e horizontal ao mesmo tempo*'.

5.3 Exercícios

5.1. Defina o predicado `último(L,U)`, que determina o último item `U` numa lista `L`. Por exemplo, `último([a,b,c],U)`, resulta em `U=c`.

5.2. Defina o predicado `tam(L,N)`, que determina o número de itens `N` existente numa lista `L`. Por exemplo, `tam([a,b,c],N)`, resulta em `N=3`.

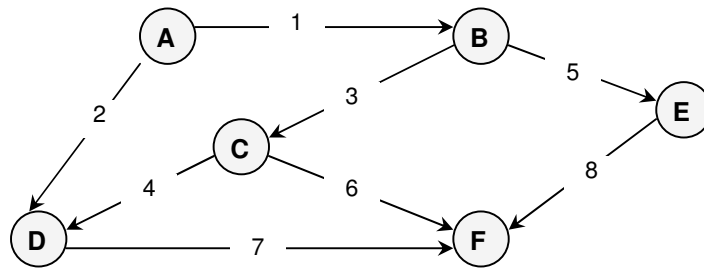
5.3. Defina o predicado `soma(L,S)` que calcula a soma `S` dos itens da lista `L`. Por exemplo, `soma([4,9,1],S)` resulta em `S=14`.

5.4. Defina o predicado `máx(L,M)` que determina o item máximo `M` na lista `L`. Por exemplo, `máx[4,9,1],M)` resulta em `M=9`.

5.5. Usando o predicado `anexa`, defina o predicado `inv(L,R)` que inverte a lista `L`. Por exemplo, `inv([b, c, a], R)` resulta em `R=[a,c,b]`.

5.6. Usando o predicado `inv`, defina o predicado `sim(L)` que verifica se uma lista é simétrica. Por exemplo, `sim([a,r,a,r,a])` resulta em `yes`.

5.7. Usando a tabela `d(0,zero)`, `d(1,um)`, ..., `d(9,nove)`, defina o predicado `txt(D,P)` que converte uma lista de dígitos numa lista de palavras. Por exemplo, `txt([7,2,1],P)` resulta em `P=[sete,dois,um]`.

5.8. O grafo a seguir representa um mapa, cujas cidades são representadas por letras e cujas estradas são representados por números.

a) Usando o predicado `estrada(Número,Origem,Destino)`, crie um programa para representar esse mapa.

b) Defina o predicado `rota(A,B,R)`, que determina uma rota `R` (lista de estradas) que leva da cidade `A` até a cidade `B`.

5.9. Um retângulo é representado pela estrutura `retângulo(A,B,C,D)`, cujos vértices são `A`, `B`, `C` e `D`, nesta ordem.

a) Defina o predicado `regular(R)` que resulta em `yes` apenas se `R` for um retângulo cujos lados sejam verticais e horizontais.

b) Defina o predicado `quadrado(R)` que resulta em `yes` apenas se `R` for um retângulo cujos lados têm as mesmas medidas.

Capítulo 6
Base de Dados Dinâmica

Base de dados dinâmica é um recurso bastante útil para implementar lógica não-monotônica e programas que adquirem conhecimento dinamicamente.

6.1 Manipulação da base de dados dinâmica

De acordo com o modelo relacional, uma *base de dados* é a especificação de um conjunto de relações. Neste sentido, um programa em Prolog é uma base de dados em que a especificação das relações é parcialmente explícita (fatos) e parcialmente implícita (regras).

Programa 6.1: *Jogadores e esportes.*

```
joga(pelé,futebol).
joga(guga,tênis).
esporte(X) :- joga(_,X).
```

Para listar as cláusulas da base de dados, usamos o predicado `listing`. Por exemplo, supondo que o Programa 6.1 tenha sido carregado na base, podemos listar as cláusulas do predicado `joga` através da seguinte consulta:

```
?- listing(joga).
joga(pelé,futebol).
joga(guga,tênis).
```

Para adicionar uma nova cláusula à base de dados podemos usar o predicado `asserta` ou `assertz`. A consulta a seguir mostra a diferença entre eles:

```
?- assertz(joga(oscar,basquete)).
Yes
?- asserta(joga(hortência,basquete)).
Yes

?- listing(joga).
joga(hortência,basquete).
joga(pelé,futebol).
joga(guga,tênis).
joga(oscar,basquete).
Yes
```

Para remover uma cláusula da base de dados, usamos o predicado `retract`. Esse predicado recebe uma estrutura como argumento e remove da base uma ou mais cláusulas (por retrocesso) que unificam com essa estrutura:

```
?- retract(joga(X,basquete)).
X = hortência ;
X = oscar ;
No
?- listing(joga).
joga(pelé, futebol).
joga(guga, tênis).
Yes
```

6.2 Aprendizagem por memorização

Grande parte do poder da linguagem Prolog vem da habilidade que os programas têm de se modificar a si próprios. Esta habilidade possibilita, por exemplo, a criação de programas capazes de aprender por memorização.

Programa 6.2: *Onde estou?*

```
:- dynamic estou/1.          % declara modificação dinâmica
estou(paulista).
ando(Destino) :-
  retract(estou(Origem)),
  asserta(estou(Destino)),
  format('Ando da ~w até a ~w',[Origem,Destino]).
```

Veja como o Programa 6.2 funciona:

```
?- estou(Onde).
Onde = paulista
Yes
?- ando(augusta).
Ando da paulista até a augusta
Yes
?- estou(Onde).
Onde = augusta
Yes
```

Para satisfazer o objetivo `ando(augusta)`, o sistema precisa remover o fato `estou(paulista)` e adicionar o fato `estou(augusta)`. Assim, quando a primeira consulta é feita novamente, o sistema encontra uma nova resposta.

6.3 Atualização da base de dados em disco

Como podemos observar, o Programa 6.2 implementa um tipo de raciocínio *não-monotônico*; já que as conclusões obtidas por ele mudam à medida que novos fatos são conhecidos. Porém, como as alterações causadas pela execução dos predicados `asserta` e `retract` são efetuadas apenas na memória, da próxima vez que o programa for carregado, a base de dados estará inalterada e o programa terá "esquecido" que *andou da Paulista até a Augusta*.

Para salvar em disco as alterações realizadas numa base de dados, podemos usar os predicados `tell` e `told`.

Programa 6.3: *Salvando uma base de dados em disco.*

```
salva(Predicado,Arquivo) :-
  tell(Arquivo),
  listing(Predicado),
  told.
```

Para recuperar uma base salva em disco, usamos o predicado `consult`.

6.4 Um exemplo completo: memorização de capitais

Como exemplo de aprendizagem por memorização, vamos apresentar um programa capaz de memorizar as capitais dos estados. Esse programa será composto por dois arquivos:

- `geo.pl`: contendo as cláusulas responsáveis pela iteração com o usuário;
- `geo.bd`: contendo as cláusulas da base de dados dinâmica.

No início da execução, o programa `geo.pl` carregará na memória as cláusulas existentes no arquivo `geo.dat`. No final, essas cláusulas serão salvas em disco, de modo que ela sejam preservadas para a próxima execução.

Programa 6.4: *Um programa que memoriza as capitais dos estados.*

```
:- dynamic capital/2.

geo :- carrega('geo.bd'),
       format('~n*** Memoriza capitais ***~n~n'),
       repeat,
         pergunta(E),
         responde(E),
         continua(R),
       R = n,
       !,
       salva(capital,'geo.bd').
```

```
carrega(A) :-
  exists_file(A),
  consult(A)
  ;
  true.

pergunta(E) :-
  format('~nQual o estado cuja capital você quer saber? '),
  gets(E).

responde(E) :-
  capital(C, E),
  !,
  format('A capital de ~w é ~w.~n',[E,C]).

responde(E) :-
  format('Também não sei. Qual é a capital de ~w? ',[E]),
  gets(C),
  asserta(capital(C,E)).

continua(R) :-
  format('~nContinua? [s/n] '),
  get_char(R),
  get_char('\n').

gets(S) :-
  read_line_to_codes(user,C),
  name(S,C).

salva(P,A) :-
  tell(A),
  listing(P),
  told.
```

Na primeira vez que o Programa 6.4 for executado, a base de dados `geo.bd` ainda não existirá e, portanto, ele não saberá nenhuma capital. Entretanto, à medida que o usuário for interagindo com o programa, o conhecimento do programa a respeito de capitais vai aumentado. Quando a execução terminar, a base de dados será salva em disco e, portanto, na próxima vez que o programa for executado, ele se "lembrará" de todas as capitais que aprendeu.

O Programa 6.4 faz uso de uma série de predicados extra-lógicos primitivos que são dependentes do interpretador. Maiores detalhes sobre esses predicados podem ser obtido no manual *on-line* do SWI-Prolog. Por exemplo, para obter esclarecimentos sobre o predicado name, faça a seguinte consulta:

```
?- help(name).
```

6.5 Exercícios

6.1. Supondo que a base de dados esteja inicialmente vazia, indique qual será o seu conteúdo após terem sido executadas as seguintes consultas.

```
?- asserta( metal(ferro) ).
?- assertz( metal(cobre) ).
?- asserta( metal(ouro) ).
?- assertz( metal(zinco) ).
?- retract( metal(X) ).
```

6.2. Implemente os predicados `liga`, `desliga` e `lâmpada` para que eles funcionem conforme indicado pelos exemplos a seguir:

```
?- liga, lâmpada(X).
X = acessa
Yes

?- desliga, lâmpada(X).
X = apagada
Yes
```

6.3. O predicado `asserta` adiciona um fato à base de dados, incondicionalmente, mesmo que ele já esteja lá. Para impedir essa redundância, defina o predicado `memorize`, tal que ele seja semelhante a `asserta`, mas só adicione à base de dados fatos inéditos.

6.4. Suponha um robô capaz de *andar* até um certo local e *pegar* ou *soltar* objetos. Além disso, suponha que esse robô mantém numa base de dados sua posição corrente e as respectivas posições de uma série de objetos. Implemente os predicados `pos(Obj,Loc)`, `ande(Dest)`, `pegue(Obj)` e `solte(Obj)`, de modo que o comportamento desse robô possa ser simulado, conforme exemplificado a seguir:

```
?- pos(O,L).
O = robô
L = garagem ;
O = tv
L = sala ;
No
?- pegue(tv), ande(quarto), solte(tv), ande(cozinha).
anda de garagem até sala
pega tv
anda de sala até quarto
solta tv
anda de quarto até cozinha
Yes
```

6.5. Modifique o programa desenvolvido no exercício anterior de modo que, quando for solicitado ao robô pegar um objeto cuja posição é desconhecida, ele pergunte ao usuário onde está esse objeto e atualize a sua base de dados com a nova informação. Veja um exemplo:

```
?- pos(O,L).
O = robô
L = cozinha ;
O = tv
L = quarto ;
No
?- pegue(lixo), ande(rua), solte(lixo), ande(garagem).
Onde está lixo? quintal
anda de cozinha até quintal
pega lixo
anda de quintal até rua
solta lixo
anda de rua até garagem
Yes
?- pos(O,L).
O = robô
L = garagem ;
O = lixo
L = rua ;
O = tv
L = quarto ;
No
```

6.6. Acrescente também ao programa do robô o predicado `leve(Obj,Loc)`, que leva um objeto até um determinado local. Por exemplo:

```
?- leve(tv,sala).
anda de garagem até quarto
pega tv
anda de quarto até sala
solta tv
Yes
```

Bibliografia

Mais detalhes sobre a programação em Prolog podem ser obtidos nas seguintes referências.

[1] BRATKO, I. *Prolog Programming for Artificial Intelligence*, 2nd Edition, Addison-Wesley, 1990.

[2] COVINGTON, M. A., NUTE, D. and VELLINO, A. *Prolog Programming in Depth*, 2nd Edition, Prentice-Hall, 1997.

[3] STERLING, L. and SHAPIRO, E. *The Art of Prolog - Advanced Programming Techniques*, MIT Press, 1986.